

# Scaling Bounded Model Checking By Transforming Programs With Arrays

Anushri Jana<sup>1</sup>, Uday P. Khedker<sup>2</sup>, Advaita Datar<sup>1</sup>, R. Venkatesh<sup>1</sup>, and Niyas C<sup>1</sup>

<sup>1</sup> Tata Research Development and Design Centre, Pune, India  
 {anushri.jana,advaita.datar,r.venky,niyas.c}@tcs.com

<sup>2</sup> Indian Institute of Technology Bombay, India.  
 uday@cse.iitb.ac.in

**Abstract.** Bounded Model Checking is one the most successful techniques for finding bugs in program. However, model checkers are resource hungry and are often unable to verify programs with loops iterating over large arrays. We present a transformation that enables bounded model checkers to verify a certain class of array properties. Our technique transforms an array-manipulating (ANSI-C) program to an array-free and loop-free (ANSI-C) program thereby reducing the resource requirements of a model checker significantly. Model checking of the transformed program using an off-the-shelf bounded model checker simulates the loop iterations efficiently. Thus, our transformed program is a sound abstraction of the original program and is also precise in a large number of cases—we formally characterize the class of programs for which it is guaranteed to be precise. We demonstrate the applicability and usefulness of our technique on both industry code as well as academic benchmarks.

**Keywords:** Program Transformation, Bounded Model Checking, Array, Verification.

## 1 Introduction

Bounded Model Checking is one of the most successful techniques for finding bugs [11] as evidenced by success achieved by the tools implementing this technique in verification competitions [1,2]. Given a program  $P$  and a property  $\phi$ , Bounded Model Checkers (BMCs) unroll the loops in  $P$ , a fixed number of times and search for violations to  $\phi$  in the unrolled program. However, for programs with loops of large or unknown bounds, bounded model checking instances often exceed the limits of resources available. In our experience, programs manipulating large arrays invariably have such loops iterating over indices of the array. Consequently, BMCs routinely face the issue of scalability in proving properties on arrays. The situation is no different even when the property is an *array invariant* i.e., it holds for every element of the array—a characteristic which can potentially be exploited for efficient bounded model checking.

Consider the example in Figure 1 manipulating an array of structures  $a$ . The structure has two fields,  $p$  and  $q$ , whose values are assigned in the first *for* loop (lines 8–13) such that  $a[i].q$  is the square of  $a[i].p$  for every index  $i$ . The second *for* loop (lines 14–17) asserts that this property indeed holds for each element in  $a$ . This is a *safe* program i.e.,

```

1. struct S {
2.     unsigned int p;
3.     unsigned int q;
4. } a[100000];
5. int i,k;

06. main()
07. {
08.     for(i=0; i<100000; i++)
09.     {
10.         k = i;
11.         a[i].p = k;
12.         a[i].q = k * k ;
13.     }

14.     for (i=0; i<100000; i++)
15.     {
16.         assert(a[i].q ==
17.             a[i].p * a[i].p);
18.     }

```

Fig. 1: Motivating Example

```

1. struct S{
2.     unsigned int p;
3.     unsigned int q;
4. }x_a;
5. int i_a;
6. int i,k;
7. main()
8. {
9.     i_a = nd(0,99999);

//first loop body
10. k = nd(0,100000);
11. i = i_a;
12. k = i;
13. (i == i_a)? x_a.p = k : k;
14. (i == i_a)? x_a.q = k * k : k*k ;
15. k = nd(0,100000);

//second loop body
16. i = i_a;
17. assert(((i==i_a)?x_a.q:nd())
18.     ==((i==i_a)?x_a.p:nd())
19.     *((i==i_a)?x_a.p:nd()));

```

Fig. 2: Transformed Code

the assertion does not admit a counterexample. CBMC [9], a bounded model checker for C, tries to unwind the first loop 100000 times and runs out of memory before reaching the loop with the assertion. We tried this example with several model checkers<sup>3</sup> and none of them was able to prove this property because of a large loop bound.

One of the ways of proving this example safe is to show that the property holds for any arbitrary element of the array, say at index  $i_c$ . This allows us to get rid of those parts of the program that do not update  $a[i_c]$  which, in turn, eliminates the loop iterating over all the array indices. This enables CBMC to verify the assertion without getting stuck in the loop unrolling. Moreover, since  $i_c$  is chosen nondeterministically from the indices of  $a$ , the property holds for every array element without loss of generality.

This paper presents the transformation sketched above with the aim that the transformed program is easier for a BMC to verify as compared to the original program. The transformation is over-approximative i.e., it give more values than that by the original program. This ensures that if the original program is safe with respect to the chosen property, so is the transformed program. However, the over-approximation raises two important questions spanning practical and intellectual considerations:

- 1) *Is the proposed approach practically useful? Does the transformation enable a BMC to verify real-world programs or academic benchmarks fairly often?*

We answer this through an extensive experimental evaluation over industry code as well as examples in the array category of SV-COMP 2016 benchmarks. Our approach helps CBMC to scale in each case. We further demonstrate the applicability of our technique to successfully identify a large number of false warnings (on an average 73%) reported by a static analyzer on arrays in large programs.

<sup>3</sup> Result for *motivatingExample.c* at <https://sites.google.com/site/datastructureabstraction/>

2) *Is it possible to characterize a class of properties for which it is precise?*

In order to address this we provide a formal characterization of properties for which the transformation is precise i.e., we state criteria under which the transformed program is unsafe only when the original program is unsafe (Section 6).

To summarize, this paper makes the following contributions:

- A new technique combining static analysis and model checking enabling verification of array invariant properties in programs with loops iterating over large arrays.
- A novel concept of using pair of a *witness variable* and a *witness index* which allows us to remove the loops and arrays from programs and simulate the iterations accessing different elements of arrays during model checking.
- A formal characterization of properties for which the transformation is precise.
- A transformation engine implementing the technique.
- An extensive experimental evaluation showing the applicability of our technique to real-world programs as well as academic benchmarks.

The rest of the paper starts with an informal description of the transformation (Section 2) before we define the semantics (Section 3) and formally state the transformations rules (Section 4). Section 5 and 6, respectively, describe the soundness and precision of our approach. Section 9 presents the experimental setup and results. We discuss the related work in Section 10 before concluding in Section 11.

## 2 Informal Description

Given a program  $P$  containing loops iterating over an array  $a$ , we transform it to a program  $P'$  that has a pair  $\langle x_a, i_a \rangle$  of a *witness variable* and a *witness index* for the array and its index such that  $x_a$  represents the element  $a[i_a]$  of the original program. Further, loops are replaced by their customized bodies that operate only on the witness variable  $x_a$  instead of all elements of the array  $a$ .

To understand the intuition behind our transformation, consider a trace  $t$  of  $P$  ending on the assertion  $A_n$ . Consider the last occurrence of a statement  $s : a[e_1] = e_2$  in  $t$ . We wish to transform  $P$  such that there exists a trace  $t'$  of  $P'$  ending on  $A_n$  in which the value of  $i_a$  is equal to that of  $e_1$  and the value of  $x_a$  is equal to that of  $e_2$ . We achieve this by transforming the program such that:

- $i_a$  gets a non-deterministically chosen value at the start of the program (this facilitates an arbitrary choice of array element  $a[i_a]$ ).
- array writes and reads for  $a[i_a]$  are replaced by the witness variable  $x_a$ .
- array writes other than  $a[i_a]$  are eliminated and reads are replaced by a non-deterministically chosen value.
- loop body is executed only once either unconditionally or non-deterministically; based on loop characteristics. During the execution of the transformed loop body,
  - the loop iterator variable either gets the value of  $i_a$  or a non-deterministically chosen value (depending on loop characteristics), and
  - all other scalar variables whose values may be different in different iterations gets non-deterministically chosen values.

Figure 2 shows the transformed program  $P'$  for the program  $P$  of Figure 1. Function  $\text{nd}(l, u)$  returns a non-deterministically chosen value in the range  $[l..u]$ . In  $P'$ , the witness index  $i_a$  for array  $a$  is globally assigned a non-deterministically chosen value within the range of array size (at line 9). In a run of BMC, the assertion is checked for this non-deterministically chosen element  $a[i_a]$ . To ensure that the values for the same index  $a[i_a]$  are written and read, we replace the array accesses by the witness variable  $x_a$  only when the value of the index  $i$  matches with that of  $i_a$  (lines 13, 14 and 17). We remove the loop header but retain the loop body. To over-approximate the effect of the removal of loop iterations, we add non-deterministic assignments to all variables modified in the loop body, at the start of the transformed loop body and also after the transformed loop body (lines 11 and 15). Note that we retain the original assignment statements too (line 12). Since the loops at line 8 and line 14 in the original program iterate over the entire array, we equate loop iterator variable  $i$  to  $i_a$  (line 11 and 16) and the transformed loop bodies (lines 10–14 and lines 16–17) are executed unconditionally.

We are using a single variable,  $x_a$ , to represent the array  $a$ . However,  $x_a$  takes different values in different runs of the BMC based on an arbitrarily chosen value of  $i_a$  and  $k$ . Our technique is able to verify the program in Figure 1 because we do not conflate the values of expressions across different runs as is done in any static analysis. In a sense, we go beyond a static analysis and use a dynamic analysis through a BMC to verify the property.

We explain the transformation rules formally in Section 4. The transformed program can be verified by an off-the-shelf BMC. Note that each index will be considered in some run of the BMC since  $i_a$  is chosen non-deterministically. Hence, if an assertion fails for any index in the original program, it fails in the transformed program too.

### 3 Semantics

In this section we formalize our technique by explaining the language and defining a representation of states.

#### 3.1 Language

We formulate our analysis over a language modelled on C. For simplicity of exposition we restrict our description to a subset of C which includes structures and 1-dimensional arrays. For a given program, let  $\mathbb{C}$ ,  $\mathbb{V}$ , and  $\mathbb{E}$  be the sets of values computed by the program, the variables appearing in the program, and the expressions appearing in the program, respectively. A value  $c \in \mathbb{C}$  can be an integer, a floating-point or a boolean value. A variable  $v \in \mathbb{V}$  can be a scalar variable, a structure variable, or an array variable. We define our program to have only one array variable denoted as  $a$ . However, our implementation handles multiple arrays as explained in our technical report [?]. We also define  $\mathbb{E}_A \subseteq \mathbb{E}$  as the set of array expressions of the form  $a[E]$ . An lval  $L$  can be an array access expression or a variable. Let  $c \in \mathbb{C}$ ,  $x, i \in (\mathbb{V} - \{a\})$ . We consider assignment statements, conditional statements, loop statements, and assertion statements defined by the following grammar. We define the grammar of our language using the

following non-terminals: Program  $P$  consists of statements  $S$  which may use lvalues  $L$  and expressions  $E$ . We assume that programs are type correct as per C typing rules.

$$\begin{aligned}
P &\rightarrow S \\
S &\rightarrow \text{if } (E) S \text{ else } S \mid \text{if } (E) S \mid \text{for } (i = E; E; E) S \mid \\
&\quad S; S \mid L = E \mid \text{assert}(E) \\
L &\rightarrow a[E] \mid x \\
E &\rightarrow E \oplus E \mid L \mid c
\end{aligned} \tag{1}$$

In practice, we analyze ANSI-C language programs that include functions, pointers, composite data-structures, all kinds of definitions, and all control structures except multi-dimensional arrays.

### 3.2 Representing Program States

We define program states in terms of memory locations and the values stored in memory locations. We distinguish between *atomic* variables (such as scalar and structure variables) whose values can be copied atomically using a single assignment operation, from non-atomic variables such as arrays. Since we are considering 1-dimensional arrays, the array elements are atomic locations.

Function  $\ell(a[i])$  returns the memory location corresponding to the  $i^{th}$  index of array  $a$ . The memory of an input program consists of all atomic locations:

$$\mathbb{M} = (\mathbb{V} - \{a\}) \cup \{\ell(a[i]) \mid 0 \leq i \leq \text{lastof}(a)\} \tag{2}$$

The function  $\text{lastof}(a)$  returns the highest index value for array  $a$ .

A *program state* is a map  $\sigma : \mathbb{M} \rightarrow \mathbb{C}$ .  $\llbracket e \rrbracket_\sigma$  denotes the value of expression  $e$  in the program state  $\sigma$ .

We transform a program by creating a pair  $\langle i_a, x_a \rangle$  for array  $a$  where  $i_a$  is the witness index and  $x_a$  is the witness variable. The memory of a transformed program with additional variables is:

$$\mathbb{M}' = (\mathbb{V} - \{a\}) \cup \{x_a\} \cup \{i_a\} \tag{3}$$

For a transformed program, a program state is denoted by  $\sigma'$  and is defined over  $\mathbb{M}'$ . A set of states is denoted by  $\Sigma'$  and set of states at program point  $l$  as  $\Sigma'_l$ .  $\llbracket e \rrbracket_{\Sigma'}$  denotes the set of value of expression  $e$  in the set of program states  $\Sigma'$ .

We illustrate the states in the original and the transformed program through an example. Let a program  $P$  have an array variable  $a$  and a variable  $k$  holding the size of the array  $a$ . Let the array contain the values  $c_i \in \mathbb{C}$ ,  $0 \leq i < n$ , where  $n \in \mathbb{C}$  is the value of size of the array. Then, a program state,  $\sigma$  at any program point  $l$  can be:

$$\sigma = \{(k, n), (\ell(a[0]), c_0), (\ell(a[1]), c_1), \dots, (\ell(a[n-1]), c_{n-1})\} \tag{4}$$

In the transformed program  $P'$ , let  $x_a$  and  $i_a$  be the witness variable and witness index respectively. Let  $l'$  be the program point in  $P'$  that corresponds to  $l$  in  $P$ . Then, all possible states in the transformed program at  $l'$  are,

$$\begin{aligned}
\sigma'_0 &= \{(k, n), (i_a, 0), (x_a, c_0)\} \\
\sigma'_1 &= \{(k, n), (i_a, 1), (x_a, c_1)\} \\
&\dots \\
\sigma'_{n-1} &= \{(k, n), (i_a, n-1), (x_a, c_{n-1})\} \\
\Sigma' &= \{\sigma'_0, \sigma'_1 \dots \sigma'_{n-1}\}
\end{aligned}$$

We now formally define how the states at a program point in the transformed program represents a state at the corresponding program point in the original program.

**Definition 1.** Let  $\sigma$  be a state at a program point in  $P$  and let  $\sigma'$  be a state at the corresponding program point in  $P'$ . Then, for any index  $c_1$  under consideration,  $\sigma'$  represents  $\sigma$ , denoted as  $\sigma' \rightsquigarrow \sigma$ , if

$$(\ell(a[c_1]), c_2) \in \sigma \Rightarrow \sigma' = \{(i_a, c_1), (x_a, c_2)\} \cup \{(y, c) \mid (y, c) \in \sigma, y \in (\mathbb{V} - \{a\})\}$$

**Definition 2.** Let  $\sigma$  be a state at a program point in  $P$  and let  $\Sigma'$  be set of states at the corresponding program point in  $P'$ . Then,  $\Sigma'$  represents  $\sigma$ , denoted as  $\Sigma' \rightsquigarrow \sigma$ , if

$$\forall c_i \text{ such that } (\ell(a[c_i]), c_j) \in \sigma, \exists \sigma' \in \Sigma' \text{ such that } \sigma' \rightsquigarrow \sigma.$$

Let  $A_n$  be the assertion at line  $n$  in program  $P$ . Let  $\sigma$  be a state reaching  $A_n$  in the original program with pair  $(\ell(a[\llbracket e_1 \rrbracket_\sigma]), \llbracket e_2 \rrbracket_\sigma)$ . Let  $\sigma'$  be the state in transformed program,  $\sigma'$  represents  $\sigma$ . Thus,  $\sigma'$  has two pairs,  $(i_a, \llbracket e_3 \rrbracket_{\sigma'})$  and  $(x_a, \llbracket e_4 \rrbracket_{\sigma'})$  such that  $\llbracket e_3 \rrbracket_{\sigma'} = \llbracket e_1 \rrbracket_\sigma$  and  $\llbracket e_4 \rrbracket_{\sigma'} = \llbracket e_2 \rrbracket_\sigma$ . Hence, if the assertion  $A_n$  holds in transformed program, it holds in the original program too.

## 4 Transformation

The transformation rules are given in Figure 3. A transformed program satisfies the following grammar derived from that of the original program (Grammar 1). Let  $x, x_a, i_a \in \mathbb{V}$  denote a scalar variable, the witness variable, and the witness index, respectively. Let  $c, l, u \in \mathbb{C}$  denote the values. Then,

$$\begin{aligned}
P &\rightarrow I; S \\
I &\rightarrow i_a = nd(l, u) \\
S &\rightarrow \text{if}(E) S \text{ else } S \mid \text{if}(E) S \mid S; S \mid L = E \mid \text{assert}(E) \\
L &\rightarrow x \mid x_a \mid i_a \\
E &\rightarrow E \oplus E \mid L \mid c \mid nd() \mid nd(l, u)
\end{aligned} \tag{5}$$

The non-terminal  $I$  represents the initialization statements for witness index. Witness variable is initialized in the same scope as that in the original program.

Following are the functions used in the transformation rules.

- Function  $nd(l, u)$  returns a non-deterministically chosen value between the lower limit  $l$  and the upper limit  $u$ . When a range is not provided,  $nd$  returns a non-deterministically chosen value based on the type of  $L$ .

$transform(E) =$		
$E \equiv (E_1 \oplus E_2)$	$\Rightarrow emit(transform(E_1) \oplus transform(E_2))$	(3.E <sub>1</sub> )
$E \in \mathbb{E}_A, E \equiv a[E_1]$	$\Rightarrow emit((E_1 == i_a) ? x_a : nd())$	(3.E <sub>2</sub> )
otherwise	$\Rightarrow emit(E)$	(3.E <sub>3</sub> )
<hr/>		
$transform(S) =$		
$S \equiv (L = E), L \equiv a[E_1]$	$\Rightarrow emit((E_1 == i_a) ?$ $x_a = transform(E) : transform(E))$	(3.S <sub>1</sub> )
$S \equiv (L = E), L \not\equiv a[E_1]$	$\Rightarrow emit(L = transform(E))$	(3.S <sub>2</sub> )
$S \equiv (\text{for}(i = E_1; E_2; E_3) S_1),$ $fullarrayaccess(S),$ $u \in loopdefs(S_1)$	$\Rightarrow emit(u = nd(); \quad // \forall u \in loopdefs(S_1)$ $i = i_a;$ $transform(S_1);$ $u = nd(); \quad // \forall u \in loopdefs(S_1)$ $)$	(3.S <sub>3</sub> )
$S \equiv (\text{for}(i = E_1; E_2; E_3) S_1),$ $\neg fullarrayaccess(S),$ $u \in loopdefs(S_1)$	$\Rightarrow emit(\text{if}(nd(0, 1))$ $\{ u = nd(); \quad // \forall u \in loopdefs(S_1)$ $i = nd(loopbound(S));$ $transform(S_1);$ $\}$ $u = nd(); \quad // \forall u \in loopdefs(S_1)$ $)$	(3.S <sub>4</sub> )
$S \equiv (\text{if}(E) S_1 \text{ else } S_2)$	$\Rightarrow emit(\text{if}(transform(E))$ $transform(S_1) \text{ else } transform(S_2))$	(3.S <sub>5</sub> )
$S \equiv (\text{if}(E) S_1)$	$\Rightarrow emit(\text{if}(transform(E)) transform(S_1))$	(3.S <sub>6</sub> )
$S \equiv (S_1; S_2)$	$\Rightarrow emit(transform(S_1); transform(S_2))$	(3.S <sub>7</sub> )
$S \equiv (\text{assert}(E))$	$\Rightarrow emit(\text{assert}(transform(E)))$	(3.S <sub>8</sub> )
otherwise	$\Rightarrow emit(S)$	(3.S <sub>9</sub> )
<hr/>		
$transform(P) =$		
$P \equiv S$	$\Rightarrow emit(i_a = nd(lastof(a));$ $transform(S)$ $)$	(3.P)

Fig. 3: Program transformation rules. Non-terminals  $P, S, E, L$  represent the code fragment in the input program derivable from them.

- Function *transform* transforms the code represented by its argument non-terminal. Function *emit* shows the actual code that would be emitted. We assume that it takes the code emitted by *transform* and possibly some additional statements and outputs the combined code. It has been used only to distinguish the transformation time activity and run time activity. For example, the boolean conditions in cases 3.E<sub>2</sub> and 3.S<sub>1</sub> are not evaluated by the body of function *transform* but is a part of the transformed code and is evaluated at run time when the transformed program is

executed. Similar remarks apply to the **if** statements and other operations inside the parenthesis of *emit* function.

- Function *fullarrayaccess*(*S*) analyzes<sup>4</sup> the characteristics of the loop *S*.
  - When the loop *S* accesses array *a* completely, *fullarrayaccess*(*S*) returns true. This means that loop either reads from or writes to all the indices of the array; this could well be under different conditions in the code.
  - When the loop *S* accesses array *a* partially, *fullarrayaccess*(*S*) returns false. This means that the loop may not access all the indices or some indices are being read while some other indices are being written.
  - When loop *S* does not access an array, *fullarrayaccess*(*S*) returns false.
- Function *loopdefs*(*S*) returns a possibly over-approximated set of variables modified in loop *S*.
  - Scalar variables are included in this set if they appear on the left hand side of an assignment statement in *S* (except when the RHS is a constant).
  - Loop iterator variable *i* of loop *S* is not included in this set.
  - Array variable *a* is included in this set when the array access expression appears on the left hand side of an assignment and the value of index expression is different from the current value of the loop iterator *i*.
- Function *lastof*(*a*) returns the highest index value for array *a*.

With the above functions, the transformation rules are easy to understand. Here we explain non-trivial transformations.

- To choose an array index for a run, witness index ( $i_a$ ) is initialized at the start of the program to a non-deterministically chosen value from the range of the indices of the array (case 3.*P*). This value determines the array element ( $a[i_a]$ ) represented by the witness variable ( $x_a$ ).
- An array access expression in LHS or RHS is replaced by the witness variable ( $x_a$ ) only when the values of the witness index and index expression of the array access expression match. When the values do not match, it implies that an element other than  $i_a$  is being accessed. Hence for any other index the assignment does not happen (case 3.*S*<sub>1</sub>). However RHS of such assignment statement is retained to handle side-effects of expressions (not included in the grammar). Similarly, when an index other than  $i_a$  is read, it is replaced by a non-deterministically chosen value (case 3.*E*<sub>2</sub>).
- Loop iterations are eliminated by removing the loop header containing initialization, test, and increment expression for the loop iterator variable. The loop bodies are transformed as follows :
  - Each variable in the set returned by *loopdefs*(*S*) is assigned a non-deterministically chosen value at the start of the loop body and also after the loop body. These assignments ensure that values dependent on loop iterations are over-approximated when used inside or outside the loop body.
  - The loop iterator *i* is a special scalar variable. A loop *S* where *fullarrayaccess*(*S*) holds (case 3.*S*<sub>3</sub>) essentially means that loop bound is same as the array size and array is accessed using loop iterator as index. Hence it is safe to replace array access with  $x_a$  where the values of loop iterator and index expression match. To ensure this we equate loop iterator with  $i_a$ . This models the behaviour

---

<sup>4</sup> Results of analysis may be over-approximated.



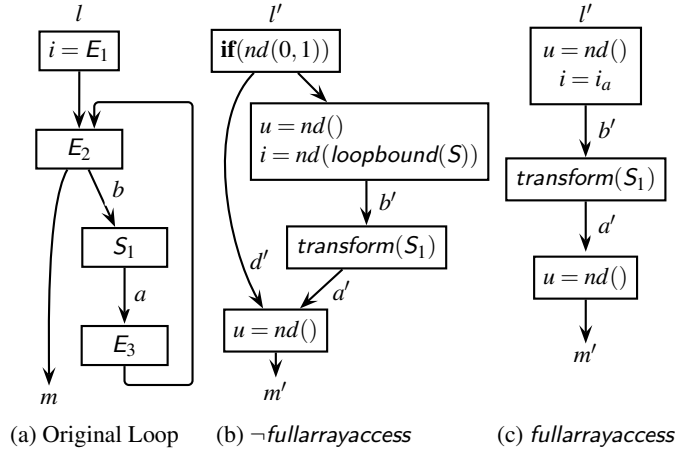


Fig. 4: Transforming a loop

of the original program precisely. However, when  $fullarrayaccess(S)$  does not hold (case 3.S<sub>4</sub>), we assign loop iterator  $i$  to a non-deterministically chosen value from the loop bound.

- Each statement in the loop body is transformed as per the transformation rules.
- Finally, the entire loop body is made conditional using a non-deterministically chosen true/false value when  $fullarrayaccess(S)$  does not hold. This models the partial accesses of array indices which imply that some of the values defined before the loop may reach after the loop. However, the transformed loop body is unconditionally executed when  $fullarrayaccess(S)$  holds.

## 5 Soundness of Abstraction

In this section, We argue that the proposed abstraction is sound, i.e. if the transformed program is safe, then so is the original program. As discussed in Section 3, the soundness of abstraction is immediate if the abstract states “represent” the original states. We, therefore, prove that the proposed transformation ensures that the *represents* relation,  $\rightsquigarrow$ , holds between abstract and original states. The proof is by induction on the number of transformation rules applied. For the base case, we prove that  $\rightsquigarrow$  holds in the beginning - before applying any transformation (lemma 1). In the inductive step, we prove that if  $\rightsquigarrow$  holds at some stage during the abstraction, then the subsequent transformation continues to preserve  $\rightsquigarrow$ . We prove this by structural induction on program transformations. Lemma 3 helps us in establishing the base case of this structural induction.

**Lemma 1.** *Let the start of the original program (i.e. the program point just before the code derivable from non-terminal  $S$  in production  $P \rightarrow S$  in grammar defined in equation( 1) be denoted by  $l$ . The corresponding program point in the transformed program  $P'$ , denoted by  $l'$ , is just after  $l$  and just before the non-terminal  $S$  in production  $P \rightarrow l; S$  (Grammar in equation 5). Let  $\sigma$  be the state at  $l$  and  $\Sigma'$  be the set of states at  $l'$  in  $P$  and  $P'$  respectively. Then,  $\Sigma'_{l'} \rightsquigarrow \sigma_l$ .*

*Proof.* Let  $init(x)$  denote the initial value in variable  $x \in \mathbb{V} - \{a\}$ . Similarly, let  $init(a)$  denote the initial value common to all indices in array  $a$ . Then,  $\sigma_l$  in  $P$  is:

$$\sigma_l = \{(x, init(x)) \mid (x \in \mathbb{V} - \{a\})\} \cup \{(\ell(a[i]), init(a)) \mid 0 \leq i \leq lastof(a)\}$$

Since  $l'$  appears after  $l$  in  $P'$ , the witness index variable is assigned a non-deterministic value corresponding to some index of the array (case 3. $P$  in Figure 3). All other variables (including the witness variable representing array) retain their initial values. Then there exist  $\sigma'_{l'} \in \Sigma'_{l'}$  such that,

$$\sigma'_{l'} = X \cup \{(i_a, i)\}$$

where,  $X$  contains the variables with their initial values.

$$X = \{(x, init(x)) \mid (x \in \mathbb{V} - \{a\})\} \cup \{(x_a, init(a))\}$$

It can be immediately seen that  $\Sigma'_{l'} \rightsquigarrow \sigma_l$  (as per Definition 1 and 2) for every value of  $i_a$ . Note that  $x_a$  and  $i_a$  are the witness variable and witness index respectively.

**Lemma 2.** Let  $\sigma_l$  be a state at a program point  $l$  in  $P$  and  $\Sigma'_{l'}$  be set of states at the corresponding program point  $l'$  in transformed program  $P'$ . Consider an arbitrary expression  $e \in \mathbb{E}$  just after  $l$  in original program  $P$ . Then,

$$\Sigma'_{l'} \rightsquigarrow \sigma_l \Rightarrow \llbracket transform(e) \rrbracket_{\Sigma'_{l'}} \supseteq \{\llbracket e \rrbracket_{\sigma_l}\}.$$

*Proof.* Since  $e$  is derived from  $E$  (grammar 1), we prove the lemma by structural induction on the productions for  $E$  by substituting  $L$  by its right hand side.

$$E \rightarrow E_1 \oplus E_2 \mid a[E_1] \mid x \mid c$$

For the base case, that is when  $E$  derives  $x$  or  $c$ ,  $transform(E) = E$  (case 3. $E_3$  in Figure 3) and the lemma holds trivially.

Assume it holds for an arbitrary  $E$  then

- When  $E$  derives  $a[E_1]$ , from the base case it implies that  $\llbracket transform(E_1) \rrbracket_{\Sigma'_{l'}} \supseteq \llbracket E_1 \rrbracket_{\sigma_l}$ . This means that in  $P'$  more indices are read than in the  $P$ . Hence the lemma holds.
- When  $E$  derives  $E_1 \oplus E_2$  - From the hypothesis  $\Sigma'_{l'} \rightsquigarrow \sigma_l$  it follows that  $\llbracket transform(E_1) \rrbracket_{\Sigma'_{l'}} \supseteq \llbracket E_1 \rrbracket_{\sigma_l}$  and  $\llbracket transform(E_2) \rrbracket_{\Sigma'_{l'}} \supseteq \llbracket E_2 \rrbracket_{\sigma_l}$ .

Therefore  $\llbracket transform(E) \rrbracket_{\Sigma'_{l'}} \supseteq \llbracket E \rrbracket_{\sigma_l}$ .

**Lemma 3.** Let  $l$  and  $m$  be the program points just before and after a statement  $s$  in  $P$  and let  $\sigma_l$  and  $\sigma_m$  be the states at  $l$  and  $m$  respectively. Let  $l'$  and  $m'$  be the program points just before and after the corresponding transformed statement  $transform(s)$  in  $P'$ . Let  $\Sigma'_{l'}$  and  $\Sigma'_{m'}$  be the set of states at  $l'$  and  $m'$  respectively. Then,  $\Sigma'_{l'} \rightsquigarrow \sigma_l \Rightarrow \Sigma'_{m'} \rightsquigarrow \sigma_m$ .

*Proof.* Statement  $s$  is derived from non-terminal  $S$  in the grammar 1. Hence, we prove the lemma by structural induction on  $S$ .

For the base case, consider the non-recursive productions of  $S$ :

- 1) For the right hand side ( $L = E$ ), there are two cases to consider-
  - When ( $L = E$ ) derives  $a[e_1] = e_2$  (case 3.S<sub>1</sub> in Figure 3), then since  $\text{transform}(e_2)$  always over-approximates (Lemma 2) then  $\Sigma'_{m'} \rightsquigarrow \sigma_m$  trivially.
  - When  $L$  is not an array, the  $\text{transform}(E)$  is assigned to unchanged  $L$ . Since  $\text{transform}(E)$  is always over-approximated  $\Sigma'_{m'} \rightsquigarrow \sigma_m$ .
- 2) For the right hand side **assert**( $E$ ), the states do not change and the lemma holds trivially.

For the inductive step, we consider the recursive productions of  $S$ .

- 1) For the right hand side  $S_1; S_2$ , the lemma holds by induction.
- 2) For the right hand side **if**( $E$ )  $S_1$  **else**  $S_2$ , the set of values of  $\text{transform}(E)$  is over-approximated (lemma 2) and it does not alter the state. Hence,  $\rightsquigarrow$  holds at the start of  $S_1$  and  $S_2$ . By induction, the lemma holds whether  $S_1$  is executed or  $S_2$  and hence it holds after the **if** statement.
- 3) For the right hand side **if**( $E$ )  $S_1$ , the argument is similar to the case of **if**( $E$ )  $S_1$  **else**  $S_2$ .
- 4) For the right hand side **for**( $i = E_1; E_2; E_3$ )  $S_1$ , let  $b$  and  $a$  be the program point just before and after  $S_1$  and  $b'$  and  $a'$  be the program point just before and after  $\text{transform}(S_1)$ . We transform the loop in two ways as illustrated in Figure 4. The loop body has either an assignment statement of the form,  $a[e_1] = e_2$  or an expression of the form  $a[e]$  or both.

- (a) We first show that the lemma holds when *fullarrayaccess* does not hold (case 3.S<sub>4</sub>)

In this case  $a[c]$  may not be referred to or updated, where  $(i_a, c) \in \sigma'_{l'}$ .

- First lets consider the path  $l'$  to  $d'$  to  $m'$  (see Figure 4.b). In the transformed program since any path can be chosen non-deterministically, this is the path we take when  $a[c]$  is not referred to or updated. Since in this path we add non-deterministic assignments to variables in the set  $\text{loopdefs}(S)$ ,  $\rightsquigarrow$  holds at the end of the **for** statement.
- For the other path (see Figure 4.b), which goes through the transformed loop body, we establish the lemma in following steps-
  - i.  $\sigma'_{l'} \rightsquigarrow \sigma_l \Rightarrow \sigma'_{b'} \rightsquigarrow \sigma_b$
  - ii.  $\sigma'_{b'} \rightsquigarrow \sigma_b \Rightarrow \sigma'_{a'} \rightsquigarrow \sigma_a$
  - iii.  $\sigma'_{a'} \rightsquigarrow \sigma_a \Rightarrow \sigma'_{m'} \rightsquigarrow \sigma_m$

Step-(ii) follows from induction. Step-(iii) follows a similar argument as in the case above, since the variables that get modified obtain a non-deterministic value. Whenever  $a \in \text{loopdefs}(S)$  or array is not modified by  $S$ , a similar argument can be used for step - (i) too.

Now for the case when  $a[c]$  is modified by  $S$  and  $a \notin \text{loopdefs}(S)$  consider  $\sigma$  at the start of the iteration in which  $a[c]$  is modified. At this point  $\sigma'_{b'} \rightsquigarrow \sigma_b$  because all variables have been assigned non-deterministic values,  $\sigma'_{l'} \rightsquigarrow \sigma_l$  and  $a[c]$  has not been modified so far. Therefore it inductively follows  $\sigma'_{a'} \rightsquigarrow \sigma_a$  and therefore  $\sigma'_{m'} \rightsquigarrow \sigma_m$ . Since,  $\sigma'_{m'} \in \Sigma'_{m'}$  hence  $\Sigma'_{m'} \rightsquigarrow \sigma_m$ .

- (b) In the case of *fullarrayaccess*,  $a[c]$  is always modified by  $S$ . We can therefore use the same argument as in the case when  $a[c]$  is modified, above.

Hence the lemma.

**Theorem 1.** *If the assertion  $A_n$  is violated in the original program  $P$ , then it will be violated in transformed program  $P'$  also.*

*Proof.* Let the assert get violated for some  $a[c]$ . Since  $i_a$  is initialized non-deterministically it can take the value  $c$  and we have shown in Lemma 2 that all expressions in  $P'$  are over-approximated. Lemma 1 and Lemma 3 ensure the premise for Lemma 2. Hence the theorem follows.

## 6 Precision

We characterize the assertions for which our transformation is precise – an assertion will fail in  $P'$  if and only if it does so in  $P$ . We denote such an assertion as  $A_n^{inv}$ . We focus on  $A_n^{inv}$  in a loop. In case if array accesses outside loops, we do not claim precision. Our experience shows that such situations are rare in programs with large arrays.

Our transformations replace array access expressions and loop statements while the statements involving scalars alone outside the loop remain unmodified. Hence precision criteria need to focus on the statements within loops and not outside it.

Let assertion  $A_n^{inv}$  appear in a loop statement  $S_{A_n^{inv}}$ . Let  $\mathbb{V}_{imp}$  be the set of variables and  $\mathbb{E}_{imp}$  be the set of array access expressions on which  $A_n^{inv}$  is data or control dependent within the loop  $S_{A_n^{inv}}$ . Let the set of loop statements from where definitions reach  $A_n^{inv}$  be denoted by  $\mathbb{S}_{def}$ , note that this set is a transitive closure for data dependence. Our technique is precise when:

- *fullarrayaccess*( $S$ ) holds for each  $S \in \{S_{A_n^{inv}}\} \cup \mathbb{S}_{def}$  (rule  $l_1$ )
- If  $a[e] \in \mathbb{E}_{imp}$  then
  - the index expression  $e = i$  where  $i$  is the loop iterator of loop  $S_{A_n^{inv}}$  (rule  $a_2$ )
  - $a \notin loopdefs(S)$  where  $S \in \{S_{A_n^{inv}}\} \cup \mathbb{S}_{def}$  (rule  $a_3$ )
- If  $x \in \mathbb{V}_{imp}$  then  $x \notin loopdefs(S)$  where  $S \in \{S_{A_n^{inv}}\} \cup \mathbb{S}_{def}$  (rule  $s_4$ )
- For an assignment statement of the form  $a[e_1] = e_2$  in loop  $S$  where  $S \in \mathbb{S}_{def}$ ,
  - if  $e_2$  is an array access expression then it must be of the form  $a[i]$  where  $i$  is the loop iterator of loop  $S$  (rule  $d_5$ )
  - if  $e_2$  is  $x$  then  $x \notin loopdefs(S)$  where  $S \in \mathbb{S}_{def}$  (rule  $d_6$ )

**Theorem 2.** *If an assertion  $A_n^{inv}$  that satisfies above rules holds in the original program  $P$ , then it will hold in the transformed program  $P'$  also.*

*Proof.* The transformed program is over-approximative because our transformation rules (3.S<sub>3</sub>, 3.S<sub>4</sub>, 3.E<sub>2</sub>) introduce non-deterministically chosen values. We prove this theorem by showing that if assertion is of the form  $A_n^{inv}$  then none of these transformation rules introduce non-deterministically chosen values in the transformed program.

- Since rule  $l_1$  holds unconditionally, loops will be transformed as per 3.S<sub>3</sub> and not 3.S<sub>4</sub> and extra paths are not added. Also, the assignment  $i = i_a$  at the start of the loop ensures that the condition is true for each transformed array access expression.
- When rule  $a_2$  holds, rule  $l_1$  also holds and  $a[e]$  is always replaced by  $x_a$  (case 3.E<sub>2</sub>).
- When rule  $a_3$  holds, assignment  $x_a = nd()$  is not added (case 3.S<sub>3</sub>).
- When rule  $s_4$  holds, assignment  $x = nd()$  is not added (case 3.S<sub>3</sub>).
- When rule  $d_5$  holds, rule  $l_1$  holds and  $a[e]$  in RHS is replaced by  $x_a$  (case 3.E<sub>2</sub>).
- When rule  $d_6$  holds, scalars used in RHS are not assigned a non-deterministically chosen values.

Rules  $s_4$  and  $d_6$  are strong requirements for ensuring that non-deterministically chosen values do not reach  $A_n^{inv}$ . We can relax these rules when  $x \in loopdefs(S_{A_n^{inv}})$  when:

- A definition of  $x$  appears before the use of  $x$  in RHS in the loop.
- Variable  $x$  is defined with a constant value or using loop iterator  $i$  only.

Since the original assignments to  $x$  are retained in the transformed loop body, assignment of  $x$  to non-deterministically chosen value ( $x = nd()$ ) gets re-defined. Also, when  $x$  is defined with a constant or  $i$  ( $i = i_a$  is added for  $S_{A_n^{inv}}$ ), its value is not over-approximated.

The assertion in Figure 1 is  $A_n^{inv}$  since it satisfies all the rules:

- $S_{A_n^{inv}}$  is the loop at line 14 containing the assertion.  $S_{def}$  contains the loop at line 8. For both these loops, rule  $l_1$  holds.
- $\mathbb{E}_{imp}$  consists of the three array access expressions at line 16. Rule  $a_2$  and  $a_3$  hold for all the three expressions.
- $\mathbb{V}_{imp}$  consist of the loop iterator  $i$  hence rule  $s_4$  holds.
- For assignments at line 11 and 12 of the lone loop in  $S_{def}$ ,
  - rule  $d_5$  holds.
  - $x$  is  $k$  which is in  $loopdefs(S)$ . But we are using relaxed form of rule  $d_6$  here.  $k$  is defined using loop iterator  $i$  at line 10. Hence  $d_6$  hold.

Since the assertion is  $A_n^{inv}$  the transformed program in Figure 2 is not over-approximated and hence BMC is able to prove the assertion.

## 7 Optimization

We have optimized the transformations such that we model the behaviour of original program as precisely as possible without impacting the soundness of our approach.

- There can be several ways in which loops do not iterate over all the indices of an array. Depending on the way loop is written we can improve the transformations instead of applying rule 3.S<sub>4</sub>. For instance, when a loop has **break** or **continue** statement, we keep the loop header such that it iterates only once. Another instance is when the increment expression of a loop is incremented monotonically but not by 1. We remove the loop header and the transformed loop body is controlled by a conditional statement, say **if** ( $i\%3$ ) for increment expression  $i + 3$ . In both the cases, we still retain additional assignments for variables in set returned by  $loopdefs(S)$  and loop iterator.

- We further gain precision by using range analysis [29]. This analysis returns the range of value (statically) that a variable can take at a program point. This range is passed as a parameter to the function returning non-deterministic value. Thus restricting the choice from the given range.

## 8 Implementation

We have implemented our transformation engine using a static analysis framework called PRISM developed at TRDDC, Pune [10,27]. Our implementation supports ANSI-C programs with 1-dimensional arrays. Structure are field sensitive:  $x.a$  and  $x.b$  are treated as separate entities. Pointers are handled using flow sensitive and context insensitive points-to analysis as implemented in PRISM. Using static analysis we populate functions such as *fullarrayaccess*( $S$ ) and *loopdefs*( $S$ ). We use information from these functions to transform the program. Range analysis provides information for *assume* statements.

Our transformation engine takes the original C program as input and generates a valid C program without array and loop. It adds proper declarations for witness variable and witness index and assume statements too.

### 8.1 Handling of Multiple Arrays

Though, we restricted our formalization to single array in a program, our tool can handle multiple arrays in a program with following enhancements:

- Function *fullarrayaccess*( $S$ ) returns true when all the arrays accessed in the loop  $S$  are of *same* sizes. This ensures that all indices of each array is accessed. When multiple arrays with *different* sizes are being read or written *fullarrayaccess*( $S$ ) conservatively returns false.
- We also add an assignment at the start of the program where witness index of each array of *same* size are equated to receive same non-deterministically chosen value. This ensures that for a loop, where *fullarrayaccess*( $S$ ) returns true, same index of each array is considered in a run of transformed program.

Consider the example in Figure 5 with multiple arrays. In the first loop (lines 5-11), elements of array  $a$  and  $b$  are assigned with value provided by the user. The second loop(line 12-15) assigns sum of each element of array  $a$  and array  $b$  to the corresponding element of array  $c$ . Third loop asserts that each index of array  $c$  contains the sum of elements at the same index of array  $a$  and  $b$ . This program is safe.

In the transformed program, shown in Figure 6, witness index  $i_a$ ,  $i_b$  and  $i_c$  are assigned with same non-deterministically chosen value at line 6 since array  $a$ ,  $b$  and  $c$  are of same size. For all three loops *fullarrayaccess*( $S$ ) hold, hence loop iterator is assigned with witness index of the arrays accessed inside the loop (line 7, 14 and 16). Other transformations are as explained in earlier sections. BMC successfully verified the transformed program.

The program in Figure 7 has multiple arrays too but with different sizes. In the transformed program as shown in Figure 8 witness index  $i_a$  and  $i_b$  are initialized at line 5 and

```

01. int a[100000], b[100000],
    c[100000];
02. int i, v;

03. main()
04. {
/* Loop1 */
05. for(i = 0; i < SIZE; i++)
06. {
07.   v = input();
08.   a[i] = v;
09.   v = input();
10.   b[i] = v;
11. }
/* Loop2 */
12. for(i = 0; i < 100000; i++)
13. {
14.   c[i] = a[i] + b[i];
15. }
/* Loop3 */
16. for(i = 0; i < 100000; i++)
17. {
18.   assert(c[i] == a[i] + b[i]);
19. }
20. }

```

Fig. 5: Multiple Array - Program 1

```

01. int x_a, x_b, x_c;
02. int i_a, i_b, i_c;
03. int i, v;

04. main()
05. {
06.   i_a=i_b=i_c=nd(0,100000);
/* Loop1 body */
07.   i=i_a; i=i_b;
08.   v = nd();
09.   v = input();
10.   (i==i_a)? x_a= v : v;
11.   v = input();
12.   (i==i_b)? x_b= v : v;
13.   v = nd();
/* Loop2 body */
14.   i=i_a; i=i_b; i=i_c;
15.   (i==i_c)? x_c=((i==i_a)?x_a:nd()
    +(i==i_b)?x_b:nd()):
    ((i==i_a)?x_a:nd()
    +(i==i_b)?x_b:nd());
/* Loop3 body */
16.   i=i_a; i=i_b; i=i_c;
17.   assert(i==i_c)? x_c
    ==((i==i_a)?x_a:nd()
    +(i==i_b)?x_b:nd()
    :((i==i_a)?x_a:nd()
    +(i==i_b)?x_b:nd());
18. }

```

Fig. 6: Transformed Program

6 to a non-deterministically chosen value from their respective range of the indices In *Loop1* *fullarrayaccess* (*S*) does not hold since different sized arrays are accessed. Since the transformed program is over-approximated, BMC could not prove this assertion. However, our approach for multiple arrays is sound in all scenarios.

## 9 Experimental Evaluation

The experiments are performed on a 64-bit Linux machine with 16 Intel Xeon processors running at 2.4GHz, and 20GB of RAM. Although we could take any off-the-shelf BMC for C program to verify the transformed code, we use CBMC in our experiments as it is known to handle all the constructs of ANSI-C. We performed experiments under three setups- (i) SV-COMP benchmarks (ii) 3 large real-life applications (iii) a set of small programs based on the patterns observed in these applications.

### 9.1 Setup 1 : SV-COMP Benchmarks

SV-COMP benchmarks [33] contain an established set of programs under various categories intended for comparing software verifiers. Results for *ArraysReach*<sup>5</sup> from the *array* category for CBMC used in SV-COMP 2016 (*CBMC<sub>α</sub>*), CBMC 5.4 (*CBMC<sub>β</sub>*)

<sup>5</sup> Programs in *ArrayMemSafety* access arrays without using index and cannot be transformed.

```

01. int i,x = 0,y = 0;
02. int a[100000],b[50000];

03. main()
04. {
05.   x = user_input();
06.   y = user_input();
/* Loop1 */
07.   for(i = 0 ; i< 50000; i++)
08.   {
09.     a[i] = x;
10.     b[i] = y;
11.     a[i+50000] = x*2;
12.     x++; y++;
13.   }
/* Loop2 */
14.   for(i=0; i < 50000; i++)
15.   {
16.     assert(a[i]*2==a[i+50000]);
17.   }
18. }

```

Fig. 7: Multiple Array - Program 2

```

01. int i_a, i_b;
02. int i, x_a, x_b, x =0, y=0;

03. main()
04. {
05.   i_a = nd(0,99999);
06.   i_b = nd(0,49999);
07.   x = user_input();
08.   y = user_input();
/* Loop1 body */
09.   if(nd(0,1))
10.   {
11.     x = nd(); y = nd();
12.     i = nd(0,49999);
13.     (i==i_a)?x_a = x:x;
14.     (i==i_b)?x_b = y:y;
15.     (i+50000==i_a)?x_a=x*2:x*2;
16.     x++; y++;
17.   }
18.   x = nd(); y = nd();
/* Loop2 body */
19.   i = nd(0,49999);
20.   assert((i==i_x)?(x_a*2):nd()==
          (i+50000==i_x)?(x_a):nd());
21. }

```

Fig. 8: Transformed Program

Table 1: Results on SV-COMP Benchmark Programs.

#programs = 118	#correct true	#correct false	#incorrect true	#incorrect false	#no result
Expected Results	84	34	-	-	0
$CBMC_{\alpha}$	47	06	06	0	59
$CBMC_{\beta}$	9	5	0	0	104
Transformation+ $CBMC_{\beta}$	25	34	0	59	0
$CBMC_{\alpha}$ - SV-COMP2016 (unsound) CBMC, $CBMC_{\beta}$ - sound CBMC 5.4					

and CBMC 5.4 on transformed programs (Transformation+ $CBMC_{\beta}$ ) are consolidated<sup>6</sup> in Table 1. *ArraysReach* has 118 programs.  $CBMC_{\alpha}$ , an unsound version of CBMC, gave correct results for 53 programs. However,  $CBMC_{\beta}$  gave correct results for 14 programs. We compare the results of Transformation+ $CBMC_{\beta}$  on three criteria:

- Scalability: it scaled up for all 118 programs.
- Soundness: it gave sound results for all 118 programs. For the 6 programs for which  $CBMC_{\alpha}$  gave unsound results, our results are not only sound but are also precise.
- Precision: it gave precise results for 59 programs. Out of these  $CBMC_{\alpha}$  ran out of memory for 45 programs ( $CBMC_{\alpha}$  ran out of memory for 14 additional programs). On the other hand, 22 true programs reported correctly by  $CBMC_{\alpha}$  were verified as

<sup>6</sup> Case by case results available at

<https://sites.google.com/site/datastructureabstraction/home/sv-comp-benchmark-evaluation-1>



Table 2: Real-life Application Evaluation

Application details				Sliced+CBMC			Sliced +Transformation +CBMC			% False Positive Reduction
Name	Size (LoC)	$\%loop^{full}$	#Asserts	#P	#F	#T	#P	#F	#T	
<i>navi1</i>	1.54M	100	63	0	0	63	52	1	10	82.5
<i>navi2</i>	3.3M	93.4	103	0	0	103	95	1	7	92.2
<i>icecast_2.3.1</i>	336K	59.1	114	0	0	114	53	61	0	46.5
$loop^{full}$ - loop $S$ where $fullarrayaccess(S)$ holds, P - Assertion Proved, F - Assertion Failed, T - Timeout										

false by Transformation+CBMC <sub>$\beta$</sub> . Transformation+CBMC <sub>$\beta$</sub>  verified 25 programs as true which did not include 8 of the programs reported correctly as true by CBMC <sub>$\beta$</sub> .

Our technique is imprecise for the other 59 of 118 programs as they do not comply with the characterization of precision provided in Section 6. As can be seen, there is a trade-off between scalability and precision. From the view point of reliability of results, soundness is the most desirable property of a verifier. Our technique satisfies this requirement. Further, it not only scales up but is also precise in many situations implying its practical usefulness.

## 9.2 Setup 2 : Real-life Applications

We applied our technique on 3 real-life applications - *navi1* and *navi2* are industry codes implementing the navigation system of an automobile and *icecast\_2.3.1* is an open source project for streaming media [26]. To verify a meaningful property, we used *Null Pointer Dereference* (NPD) warnings generated by a sound static analysis tool build using PRISM<sup>7</sup>. PRISM performs weak updates for arrays (similar to array smashing) and hence generates a large number of warnings on arrays, most of which are false. We appended assertions only for an array or an array member dereferencing as follows. Lets say the dereference expression is  $*a[i].p$ . A statement  $assert(a[i].p! = null)$  is added in the code just before the statement containing dereference expression. We then slice<sup>8</sup> these programs as per assertion.

Table 2 shows the consolidated results of our experiments. CBMC did not scale on the sliced programs. However, after transformation (sliced+transformation+CBMC) CBMC proved 200 out of 280 assertions, taking 12 minutes on average for transformation+verification. This is much less in comparison to the time given to CBMC for sliced programs (sliced+CBMC), which was 30 minutes.

To verify the correctness of our implementation, we analyzed the warnings manually. We found that all 280 warnings were false, implying that all the assertions should have been proved successfully.

<sup>7</sup> TCS Embedded Code Analyzer (TCS ECA)

[http://www.tcs.com/offering/engineering\\_services/Pages/TCS-Embedded-Code-Analyzer.aspx](http://www.tcs.com/offering/engineering_services/Pages/TCS-Embedded-Code-Analyzer.aspx)

<sup>8</sup> PRISM implements [25] for slicing.

- Scalability: CBMC could scale up for such large applications because there are no loops in transformed programs. However, CBMC could not scale for 17 cases even after transformation because of long recursive call chain through function pointers.
- Precision: Our technique proved 200 assertions, where all the conditions for precision mentioned in Section 6 get fulfilled. In all these cases since *fullarrayaccess(S)* hold, the witness variable gets precise value in each run of BMC for each value of witness index. 63 of the assertions could not be proved since array definitions reaching at the assertion were from the loops where *fullarrayaccess(S)* did not hold and values received by witness variable is over-approximative.

Replacing array expressions with witness variable enables elimination of loops. However, by using witness index BMC simulates loop iterations with providing run-based value to witness variable. Note that the number of false warnings eliminated in an application is proportional to the number of loops for which *fullarrayaccess(S)* hold. Over a diverse set of applications, we found that our technique could eliminate 40-90% of false warnings. This is a significant value addition to static analysis tools that try to find defects and end up generating a large number of warnings.

### 9.3 Setup 3 : Our Benchmarks

Table 3: Results of various BMCs on Benchmark Programs.

Name	Expected Result	CBMC		LLBMC		SMACK+ CORRAL		ESBMC		CPAChecker	
		OP	TP	OP	TP	OP	TP	OP	TP	OP	TP
motivatingExample	S	O	S	T	S	T	S	O	S	T	S
Relation_Mem	S	T	S	T	S	T	S	T	S	T	S
break_safe	S	O	S	T	S	T	S	O	S	T	S
Nested_Struct	S	T	S	T	S	T	S	O	S	T	S
partial_lesser_bound	S	T	U	T	U	O	U	T	U	T	U
OP-Original Program, TP-Transformed Program S-Safe U-Unsafe T-Time out O-Out of memory											

Since the industry code cannot be shared, we created a set of programs based on the patterns observed in real-life applications. These benchmark programs are provided at <https://sites.google.com/site/datastructureabstraction/>.

We ran following model checkers that implement different techniques: (i) CBMC 5.4, a bounded model checker for C, (ii) LLBMC 2013.1, a low-level bounded model checker [31], and (iii) Smack + Corral<sup>9</sup> [24], (iv) ESBMC, a context-bounded model checker<sup>10</sup> [18], and (v) CPA Checker, a tool based on lazy abstraction and interpolation [14]. Time out was set for 900 seconds.

<sup>9</sup> SV-COMP 2015 winner in *array* category

<sup>10</sup> SV-COMP 2016 winner in *array* category

All 5 programs are safe. Table 3 summarizes the result of various model checkers on the original and the transformed programs. None of the model checkers scaled on the original programs, while they scaled-up for all the transformed programs. Out of 5, 4 transformed programs are reported as safe while 1 is reported as unsafe. The 5 programs reported as safe comply to the precision criteria mentioned in Section 6, specially that the *fullarrayaccess*( $S$ ) hold for all the loops. While the 1 program is reported as unsafe due to over-approximations introduced since *fullarrayaccess*( $S$ ) did not hold.

## 10 Related Work

There is a sizeable body of work on reasoning about values of array elements. We give a brief overview of the relevant literature here.

Blanchet et al. proposed the use of abstract elements for array indices [8]. Though of limited practicability, they suggested two approaches - one that uses an abstract element for each index in the array i.e. *array expansion*, and another where a single abstract element is used for the entire array i.e. *array smashing*. Array expansion is precise but it cannot be used effectively for large arrays. Array smashing, on the other hand, allows handling arbitrary arrays efficiently, but suffers significant precision losses due to its inability to perform strong updates. As a result, it cannot be applied to prove the correctness of our example in Figure 1.

While expansion and smashing are at two extremes in their use of abstract elements, there are several “midway” approaches that are aimed at combining the benefits of both. Gopan et al. [21] proposed that those elements that are read or written could be dynamically expanded to incorporate strong updates, while the remaining elements could be smashed. This was extended to partitioning arrays into symbolic intervals spanning the entire array, and using an abstract variable to represent each such interval [23]. To avoid the exponential multiplication of array slices that prevented these techniques from scaling, Cousot et al. proposed an improvement [13] that automatically, semantically divides arrays into consecutive non-overlapping segments. Clousot [19], a tool that implements this improvement, indeed scales better than [21,23] but still runs out of resources while verifying our example in Figure 1.

Another way of partitioning an array is to exploit its semantic properties and split the elements into groups. Grouping array cells of similar properties, e.g. [30], has the advantage that it allows partitions to be non-contiguous. This is orthogonal to the work of Dillig et al. [17] where they introduce fluid updates of arrays using indexed locations along with bracketing constraints, to specify the concrete elements being updated. Cornish et al. [12] apply a program-to-program translation over the LLVM intermediate representation, followed by a scalar analysis. Although the abstraction in these approaches is expressible as a composition of our abstraction followed by further abstraction, our implementation of *fullarrayaccess*( $S$ ) guarantees an array-free and loop-free programs whenever possible. Moreover, we exploit the power of model checking to obtain a precise path-sensitive analysis.

In a recent work [32], Monniaux et al. proposed to convert array programs into array-less non-linear Horn clauses. The precision of this transformation is adjustable through a Galois connection parameterized by the number of distinguished cells, which,

however, needs to be decided manually. An analogous technique, based on Horn clauses over array variables that requires user inputs in form of pre- and post-conditions, is that of [16] proposed by De Angelis et al. This is practically infeasible for real-life programs, and therefore affects scalability. Another limitation of these approaches is their back-end solvers which cannot handle non-linear arithmetic. This reflects in their inability to work for our motivating example of Figure 1. Besides, unlike these approaches, our technique needs no manual input and successfully scales to large industry code.

Template-based methods [7,22] have been very useful in synthesizing invariants but these techniques are ultimately limited by the space of possible templates that must be searched for a good candidate. This has led to semi-automatic approaches, such as [20], where the predicates are usually suggested by the user. Our approach, in contrast, is fully automatic and proves safety by solving a bounded model checking instance instead of computing an invariant explicitly.

Verification tools based on CEGAR have been applied successfully to certain classes of programs, e.g. device drivers [6]. However, this technique is orthogonal to ours. In fact, a refinement framework in addition to our abstraction would make our technique complete too. Several other techniques have been used to scale bounded model checkers to tackle complex, real-world programs such as acceleration [28] and loop-abstraction [15]. But these techniques are not shown to be beneficial in abstracting complex data structures. Booster [4] integrates acceleration and lazy abstraction with interpolants for arrays. However, there are syntactic restrictions that limit the applicability of acceleration in general for programs handling arrays. Also, interpolation over array properties is difficult, especially since the goal is not to provide any interpolant, but one that generalizes well to an invariant [3,5].

## 11 Conclusions and Future Work

Verification of programs with loops iterating over arrays is a challenging problem because of large sizes of arrays. We have explored a middle ground between the two extremes of relying completely on dynamic approaches of using model checkers on the one hand and using completely static analysis involving complex domains and fix point computations on the other hand. Our experience shows that using static analysis to transform the program and letting the model checkers do the rest is a sweet spot that enables verification of properties of arrays using an automatic technique that is generic, sound, scalable, and reasonably precise.

Our experiments show that the effectiveness of our technique depends on the characteristics of programs and properties sought to be verified. We are able to eliminate 40-90% of false warnings from diverse applications. This is a significant value addition to static analysis that try to find defects and end up generating a large number of warnings which need to be resolved manually for safety critical applications. Our effort grew out of our own experience of such manual reviews which showed a large number of warnings to be false positives.

We plan to make our technique more precise by augmenting it with a refinement step to verify the programs that are reported as unsafe by our current technique. Finally, we wish to extend our technique on other data structures such as maps or lists.

## References

1. 2015 4th International Competition on Software Verification. <http://sv-comp.sosy-lab.org/2015/results/>. [Online; accessed 25-Nov-2016].
2. 2016 5th International Competition on Software Verification. <http://sv-comp.sosy-lab.org/2016/results/results-verified/>. [Online; accessed 25-Nov-2016].
3. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design*, 2014.
4. F. Alberti, S. Ghilardi, and N. Sharygina. Booster: An acceleration-based verification framework for array programs. In *ATVA*. Springer, 2014.
5. F. Alberti and D. Monniaux. Polyhedra to the rescue of array interpolants. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015.
6. T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, 2002.
7. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, 2007.
8. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, pages 85–108. Springer, 2002.
9. CBMC. <http://www.cprover.org/cbmc/>. [Online; accessed 25-Nov-2016].
10. B. Chimdyalwar and S. Kumar. Effective false positive filtering for evolving software. In *ISEC*, 2011.
11. F. Cooty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV*, 2001.
12. J. R. M. Cornish, G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Analyzing array manipulating programs by program transformation. In *LOPSTR*. 2014.
13. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*, volume 46, 2011.
14. CPA Checker. <http://cpachecker.sosy-lab.org/>. [Online; accessed 25-Nov-2016].
15. P. Darke, B. Chimdyalwar, R. Venkatesh, U. Shrotri, and R. Metta. Over-approximating loops to prove properties using bounded model checking. In *DATE*, 2015.
16. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. A rule-based verification strategy for array manipulating programs. *Fundamenta Informaticae*, 2015.
17. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *Programming Languages and Systems*. Springer, 2010.
18. ESBMC. <http://www.esbmc.org/>. [Online; accessed 25-Nov-2016].
19. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, 2011.
20. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *ACM SIGPLAN Notices*, volume 37, 2002.
21. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. *ACM SIGPLAN Notices*, 40(1), 2005.
22. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, 2008.
23. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*, 2008.

24. A. Haran, M. Carter, M. Emmi, A. Lal, S. Qadeer, and Z. Rakamarić. Smack+corral: A modular verifier. In *TACAS*, 2015.
25. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 1990.
26. Ice Cast. <http://icecast.org/>. [Online; accessed 25-Nov-2016].
27. S. Khare, S. Saraswat, and S. Kumar. Static program analysis of large embedded code base: an experience. In *ISEC*, 2011.
28. D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in c programs for fast counterexample detection. In *CAV*, 2013.
29. S. Kumar, B. Chimdyalwar, and U. Shrotri. Precise range analysis on large industry code. In *ESEC/FSE*, 2013.
30. J. Liu and X. Rival. Abstraction of arrays based on non contiguous partitions. In *VMCAI*, 2014.
31. LLBMC. <http://llbmc.org/>. [Online; accessed 25-Nov-2016].
32. D. Monniaux and L. Gonnord. Cell morphing: from array programs to array-free horn clauses. In *International Static Analysis Symposium*, pages 361–382. Springer, 2016.
33. SV-COMP 2016 Benchmarks. <https://sv-comp.sosy-lab.org/2016/benchmarks.php>. [Online; accessed 25-Nov-2016].